

Kotlin Primer

Ramp up

Introduction

First and almost, why did i choose Kotlin ?

- Because i get more than half of Effective Java implemented by it.
- It's a modern language with less overhead than java
- You write less and safer code

An overview of what u can expect

- Basic syntax (Functions, properties vs fields, collections, classes. ...)
- Std. lib functions (let, apply, run, with,....)
- Nullability
- Sealed & data classes
- Deconstructing
- Extensions

Basic syntax - Function Declaration

In Kotlin, in contrast to Java you specify first the **name** of the variable followed by it's **type**

```
public fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

```
fun sum(a: Int, b: Int) = a + b
```

Visibility modifier can be omitted, if it's public

Method block can be omitted if it's a simple one-liner

Basic syntax – Functions & default values

You can finally use **default values** for parameters passed into a **function** or **constructor** and you can (must) also use **named parameters**

```
public fun sum(a: Int = 10, b: Int): Int {  
    return a + b  
}
```

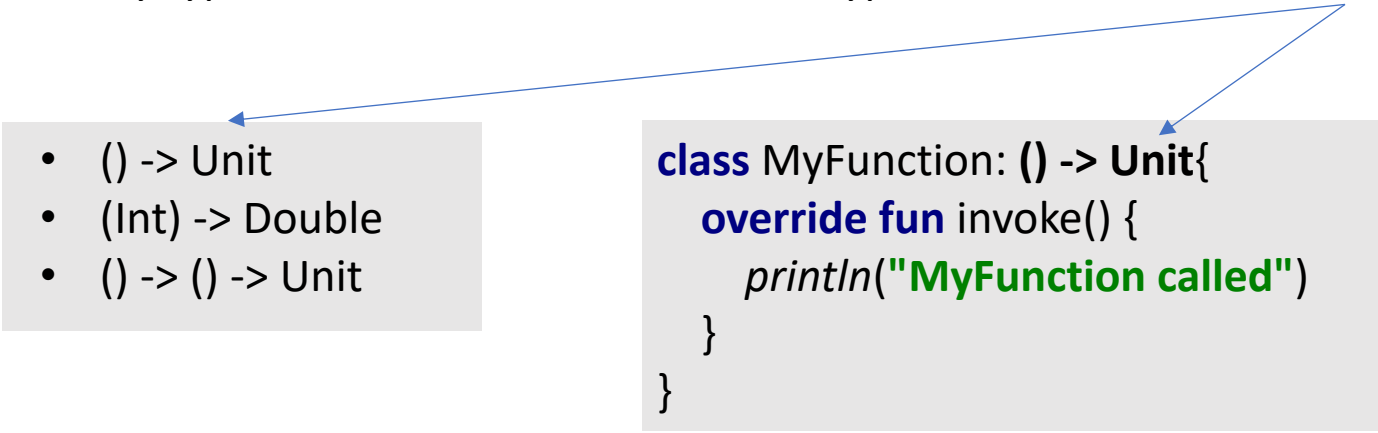
```
sum(b = 2)  
sum(a = 2, b = 5)
```

Basic syntax – Function types

In Kotlin, functions are „*first-class citizen*“, which means:

- A function can be assigned to a variable
- Passed as an argument to another function
- Returned from a function

But Kotlin is also statically typed, therefore functions have a type, which is called **function type**

- 
- () -> Unit
 - (Int) -> Double
 - () -> () -> Unit

```
class MyFunction: () -> Unit{  
    override fun invoke() {  
        println("MyFunction called")  
    }  
}
```

Basic syntax – Function literals

Another way of providing a function is to use a **function literal**. A function literal is a special notation used to simplify how a function is defined.

- **Lambda expressions**
- **Anonymous functions**

Lambda expression is a short way to define a function.

```
val greetings: () -> Unit = { println("Greetings") }
```

```
val divideByHalf: (Int) -> Int = { x -> x / 2 }
```

Anonymous function is an alternative way to define a function

```
val greetings = fun() { println("Greetings") }
```

```
val divideByHalf = fun(x: Int) = x/2
```

Basic syntax – Higher Order Functions

A higher order function is a function which takes other functions as parameter or returns a functions.

```
fun isOdd(x: Int) = x % 2 != 0
```

This function can now be used as an argument in another function in different ways:

Given the following list: `val listOfNumbers = mutableListOf<Int>(1,2,3,4)`

```
listOfNumbers.filter(::isOdd)
```

```
listOfNumbers.filter { isOdd(it) }
```

```
val predicate : (Int) -> Boolean = ::isOdd  
listOfNumbers.filter(predicate)
```

```
listOfNumbers.filter { it % 2 != 0 }
```


Basic syntax – Function composition

Consider the following function:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {  
    return { x -> f(g(x)) }  
}
```



compose(f, g) = f(g(*))

Now given the following callable references:

```
fun isOdd(x: Int) = x % 2 != 0  
fun length(string: String) = string.length
```

We can compose them by:

```
val oddLength = compose(:isOdd, ::length)
```



```
listOfStrings.filter(oddLength)
```

Basic syntax – Classes

A **big difference** to Java regarding **classes** in **Kotlin** is the fact that they **are final by default**.

There are 4 types of classes in Kotlin:

- Normal class
- Data class
- Sealed class
- Enum class

In contrast to Java and very similar to Swift, Kotlin distinguishes between a **primary** and **secondary** constructor

The rule is simple: Every secondary constructor has to call the primary constructor!

Basic syntax – Classes (Primary & Secondary Constructors)

```
class Test{} or class Test
```

```
class Test constructor(variable: Any)
```

```
class Test constructor(variable: Any){  
    init{  
        val instanceVariable = variable  
    }  
}
```

```
class TestClass constructor(val variable1: Any, val variable2: Any){  
  
    constructor(variable1: Any) : this(variable1, "")  
  
    constructor() : this("", "")  
  
}
```

Basic syntax – Class inheritance

As already mentioned every class in Kotlin is **final** by default, to open up a class for inheritance we have to use the **open** keyword.

```
open class Base {  
    open fun a(){}  
    fun b(){}  
}
```



```
class Derived : Base(){  
    final override fun a() {}  
}
```

The **same rules** also apply for **overriding properties**. With one exception:

Val properties can be overridden by var variables but not the otherway around.

Basic syntax – Object keyword

In Kotlin there is **no static keyword** nor are there **anonymous classes**.

Instead Kotlin introduced a new keyword **object** which serves two purposes.

- Object declaration
 - Replacement for singleton pattern
- Object expression
 - Replacement for anonymous classes

Basic syntax – Object declaration

Objects can't have any constructors, therefore you can't pass any argument to it but they can have members as normal classes.

```
object RedditService{  
    private lateinit var api : RedditAPI  
    init {  
        api = Retrofit.Builder()  
            .baseUrl(URL)  
            .build()  
            .create(RedditAPI::class.java)  
    }  
    fun getRedditPosts(title: String, sortOrder: String = "top") = api.getRedditPosts()  
}
```

Basic syntax – Object expression

Object expression is a structure that creates a single instance of an object:

```
val clickPt = object {  
    var x = 10  
    var y = 10  
}
```

You will use it a lot throughout your code as this pattern is used to substitute the Java anonymous classes.

```
username.addTextChangedListener(object : TextWatcher {  
    //...  
    override fun afterTextChanged(editable: Editable) {  
        //...  
    }  
}))
```

Basic syntax – Companion object

Companion objects are a brother of the object declaration. It works the same, but it takes the name of the enclosing class.

```
class TestFragment: Fragment(){  
  
    companion object Factory {  
  
        fun getInstance(arg: Map<String,String>): TestFragment{  
            val fragment = TestFragment().apply {  
                val bundle = Bundle()  
                arguments = bundle  
            }  
            return fragment  
        }  
    }  
}
```


Basic syntax – Properties vs Fields

First and all, in Kotlin we distinguish between **mutable** (**var**) & **immutable** (**val**) variables.

So how to properties (fields) look in Java and Kotlin?

```
public String name = "Chris";
```



```
var name: String = "Chris"
```

```
private String name = "Chris";
```

```
public String getName() {  
    return name;  
}
```

```
public void setName(String name) {  
    this.name = name;  
}
```

They both look very similiar, do they?

Actually these are 2 completely different concepts.

Basic syntax – General structure of properties

That is how a property in Kotlin is defined:

```
[val | var ] <Property Name> : <Property Typ> = <Property Initialisation>  
    <getter>  
    <setter>
```

There is no need to define custom getter or setter, you can directly control the visibility or behaviour of a property while u declaring it.

```
var name: String = "Chris"  
private set
```

```
var name: String = ""  
    get() = "Chris"  
private set
```

Basic syntax – Properties and fields

Normally there are no **(backing) fields** on class level in Kotlin, only properties. But you have the possibility to define a backing field when you are working inside a **customer getter** or **setter**.

This will then look like as follows:

```
var name: String = ""  
get() = field  
set(value){  
    if (value != null){  
        field = value  
    }  
}
```

We will get back to the properties syntax when we are looking into **Delegates**

Basic syntax – Collections

- **List:**
 - `listOf(T)` or `mutableListOf(T)`
- **Set**
 - `setOf(T)` or `mutableSetOf(T)`
- **Map**
 - `mapOf(key 'to' value, key 'to' value,...)` or `mutableMapOf(..)`

You can access elements inside collections by the use of the property syntax similar to JavaScript.

```
var mutableList = mutableListOf("1","2","3")  
println(mutableList[1])
```

End of section: Basic syntax
Any questions ?

Std. lib. Functions — or how one-liners can change the world

Before we dive into the programming concepts of kotlin lets look into some special functions provided by Kotlin which will help us in different ways.

The **standart.kt** library provide some more or less useful (extension) functions, most of these functions are just one-liners but with a great effect.